A DESIGN FOR A DISTRIBUTED-CONTROL MULTIPLE-PROCESSOR COMPUTER SYSTEM

Richard James Goodwin

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A DESIGN FOR A DISTRIBUTED-CONTROL
MULTIPLE-PROCESSOR COMPUTER SYSTEM

by

Richard James Goodwin

Thesis Advisor: G. A. Kildall

December, 1973

Approved for public release; distribution unlimited.

A Design for a Distributed-Control Multiple-Processor Computer System

by

Richard James Goodwin
Lieutenant, United States Navy
B.S., Naval Postgraduate School, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December, 1973

ABSTRACT

A tree-structured multiprocessing system design is proposed in which process communication is the primary link between processors. A hardware cluster, called a Processing Module, is proposed as the basic structural component. These modules literally "plug together" to form a system of arbitrary size. Each module has its own memory and runs its own hierarchically-structured operating system, the nucleus of which implements P. B. Hansen's communications primitives along with process creation and removal. Workload scheduling and process location are performed recursively in the system's tree structure. Multiprogramming is implemented system-wide, allowing processes to migrate away from overloaded modules. It is argued that the resulting system would be truly general-purpose and is subject to no limit on its size and consequent computing power.

ACKNOWLEDGEMENT

Grateful acknowledgement is made to Prof. Gary A. Kildall, at whose suggestion study of a "bus-structured" operating system was undertaken.

TABLE OF CONTENTS

I. INTRODUCTION

A. PROJECT DESCRIPTION

1. The Problem

This thesis project was undertaken to explore whether a parallel processing system could somehow have as its operating system nucleus a process or group of processes which would act as a data bus for all other processes in the system. Usually, communication mechanisms are centralized in one processor, but, in order to assure maximal independence of processors, it is desirable to somehow decentralize the bus function, as well. The problem is as follows: given that a general-purpose multiprocessing system can be governed by an operating system which is structured as a hierarchy of processes, is it theoretically feasible to implement the communication of processes at the very lowest level of that system and continue to maintain decentralization of control?

2. Extension of Uni-processing Concepts

From the outset, certain concepts appeared central to the rational design of operating systems, but it was not clear that their application to a structure having process communication as its core would prove fruitful. Much of what has appeared in the literature about the theory of process interaction and synchronization was written in the context of multiprogrammed single-processor operating systems. The extension of these ideas to multiprocessing poses fundamental design problems which do not occur when the objective is to keep a single processor busy and productive. For the simplest case of two processors, various *ad hoc* schemes can be devised to join them as a system with a shared memory. The problem immediately confronts the designer: which processor is to run the operating system? Or is there some graceful way to have them share this burden? Even if this complex problem is satisfactorily solved, it is quite likely that the solution

will not be applicable to three processors, not to mention thirty or three hundred. The difficulties of designing a multiple-processor operating system are partly those of assignment. (Which processors are to do what and when?) Other hurdles are memory access management and file protection. Assumptions must be made, therefore, about the physical arrangement of the system. For example, is memory to be accessible directly from each processor, or, for that matter, is a single memory the only alternative? A preliminary discussion of such design criteria is presented under OBJECTIVES OF PROPOSED SYSTEM (page 9).

3. Scope of Study

There is a danger inherent in theorizing about design. The desire to provide concrete descriptions in order to justify a given proposal can lead research in computer systems theory treacherously close to "chasing bits around." One can specify the physical structure only at the expense of generality in the discussion. Effort has been made in this study to provide a description of what is believed to be a feasible system, within the constraints of current and forthcoming technology. Alternate means of achieving the same effect exist in many sections of the proposed system and are noted wherever possible.

B. DEFINITIONS

For the purposes of this paper, the terms "multiprocessing" and "parallel processing" shall be used interchangeably to refer to simultaneous computation on two or more processors. The independent functioning of peripheral I/O devices is specifically excluded from this usage. This forcing of synonyms where some writers prefer a distinction is to provide for readability in sections which deal with both multiprocessing and multiprogramming.

The term "process" shall be used without rigorous definition. Since the concept of a computational process varies widely, the following constraints will be applied now

and elaborated upon when necessary later in the presentation:

(1) A process is distinct in that it has a name and may communicate with other processess, subject to system-wide limitations.

(2) A process can be created or removed (destroyed) by existing processes. (The distinction between the creation of a process and the activation of an already-existing one is that creation involves initialization of associated memory space and assignment of a name.)

(3) A process "exists" as a named sequence of software/hardware states in one processor or as a stored record known to the system and recallable for computation of its next state.

This third constraint on the definition of process is at variance with recent literature on the subject [Ref. 11, p. 11] with respect to a process existing on one and only one processor. It is appealing to speak of some processes as being "composed" of two or more other processes or of a process "proceeding" on more than one processor. This extension, as it turns out, is not universally feasible or desirable as a system design feature, even though such abstractions may be valuable from an analytical point of view.

C. AN ASSUMPTION

The primary assumption under which this study was conducted concerns the rapidly-declining cost and space/weight factors associated with Large-Scale Integration (LSI) circuitry. Whereas presently, massive efforts are made in order to gain a few percentage points of utilization out of a single CPU, it is assumed that in multiple-CPU systems of the future, some processors may stand idle for well more than half of the time, if for no other reason than that the cost of redesigning software will be prohibitive compared to simply adding more processors. Factors such as

reliability, versatility, maintainability and expandability are expected to become the main concerns. Processors are already being marketed [Ref. 12] which, exclusive of memory and power supply, are confined to a single LSI "chip" and cost less than $100. A computer system can be fabricated which will fit inside an attache case, plug into a standard wall outlet for power and attach to a standard "teletype" keyboard for input-output. In this context, a system having a relative multitude of processors may not be particularly expensive or bulky by today's standards. In terms of the so-called "large systems" in use at this writing, the system proposed in this study might appear preposterous. If the foregoing assumption proves correct, the proposal will more likely be a modest one, indeed.

C. OBJECTIVES OF PROPOSED SYSTEM

1. A General-purpose System

The system to be described in this paper is offered as one approach to general-purpose multiprocessing system design. Recent systems, such as ILLIAC IV and STAR-100, implement parallel processing but are something less than general-purpose. For example, ILLIAC IV dedicates 64 processing elements to the task of array processing under a single control unit, an efficient arrangement for vector manipulations of appropriate size but a limited one in the sense that the processing elements cannot be assigned to diverse processes simultaneously [Ref. 3, p. 76]. STAR-100 is a distributed system in which specialized computing stations perform the various functional tasks demanded by a user program. While more flexible than the ILLIAC IV, it is not clear that the STAR system can easily perform grid operations, wherein the calculation of one point is dependent on the values of orthogonally neighboring points [Ref. 4]. The primary constraint on the present undertaking, therefore, is that the design provide for general-purpose parallel processing, capable of enhanced

throughput for the broadest range of tasks possible.

2. The Large-Program Problem

Large computing systems are justified primarily by a relatively small number of large jobs which require the system's entire computing power in order to run at all. The introduction of multiprogramming in large uni-processor systems does not alter this determinant. Improved turnaround and enhanced utilization of the CPU are advantages which accrue to the smaller job classes, but the largest jobs appear more as "short circuits" to the multiprogrammed operating system. As a result, growth in these systems has been in the direction of larger amounts of on-line memory, particularly Random Access Memory (RAM). A more desirable solution to the "large-program" problem is to find parallel sequences in the program itself and submit these sequences to the multiprogrammed operating system. Beyond such alteration of the job itself, there seems little else that can be done except to switch to multiple processors. The pattern should be clear, though, that no matter how capable a system may be, someone will write a program that will require more than that system can handle. That is, no matter how many processors are available to a system at any given time, there exists, a priori, a program that can bog it down for hours on end. Again, multiprogramming of those several processors will not provide an escape from the situation, even though it will tend to optimize CPU utilizations for smaller jobs. From the standpoint of design theory, systems must be devised which treat processing power as a variable rather than as a fixed consequence of the design itself. With proper design development, the large-program problem can be reduced to an economic constraint, avoiding the need to abandon one design after another, simply to gain more and more processing power.

3. Process interaction

Although large programs dictate the gross processing

power of a system, they are not necessarily the only ones which demand greater generality of design. On the contrary, the manner in which only a few processes are required to interact can also determine the sufficiency of a given system. Theoretically, the sharing of resources in real time is completely analogous to the time-multiplexed sharing obtainable on a single processor. If, as pointed out earlier, the several parallel processes are not independent, an effect is encountered similar to "thrashing" in the execution of a page-fault algorithm: the successive blocking and restarting of each process in a large group of dependent processes requires a significant amount of system "overhead" as a necessary consequence of having only one processor. Nor does the use of multiple processors guarantee an improvement. The communication scheme which accomodates such dependencies must be efficient itself, else the operating system will continue to be the bottleneck.

4. System Hierarchy

E. W. Dijkstra, in his description of the "THE" Multiprogramming System [Ref. 7, p. 79], argues for a process hierarchy in which process communication is dependent on two lower levels of "primitives," tasks callable by higher-level processes (see Table I, page 12). "Level one" contains the segment controller, enabling problems of memory management to be treated as being invisible to the communications processes in "level two." Beneath the segment controller, at level zero, the processor allocation process runs, removing from all higher levels any concern as to when (or where) they will run. In the context of multiprogramming with a single processor, possibly even with two or three processors, this last aspect of Dijkstra's particular hierarchy is desirable. But, for a system with many processors, there are advantages to allowing certain higher-level processes to specify processor requirements dynamically, as in the case of dependent-element array processing. Hierarchical structuring of system processes

remains a powerful basis for design in any case, but the rearrangement of minimum-system processes, at least at the lowest levels of that hierarchy, can simplify the design transition to parallel processing. The attempt made in this study is to assign communication processes to level zero, allowing all higher processes to communicate with each other without any direct concern with how that communication is accomplished. An expected benefit of this arrangement is that system-wide scheduling can be more decentralized than would otherwise be possible.

Table I. Dijkstra's Hierarchy

Level	Tasks Assigned
0	Processor Allocation
1	Segment Controller
2	Message Interpreter
3	I/O Buffer Control
4	Independent-user Programs
5	Operator

5. Communication Primitives

Given that the lowest level processes are restricted to communications, (viz., internal communications, not system I/O), the definition of communications primitives must be considered. P. B. Hansen [Ref. 10, p. 239] has offered a group of four such primitives: **Send Message; Wait Answer; Send Answer; Wait Message.** These primitives are executed by a group of processes (software and/or hardware) which manage a pool of message buffers together with message queues for each process using the primitives. Once again, the concept is one formulated for single-CPU, multiprogrammed systems and reguires some manipulation before it can be applied to a feasible multi-CPU system. Hansen's four primitives are considered logically sufficient

for inter-process communication. One of the objectives of the proposed design is to assure that they can be solidly integrated with the multiprocessing environment.

II. DESCRIPTION OF PROPOSED SYSTEM

A. DESIGN PHILOSOPHY

1. Design Methodology

There are two design methodologies which are generally used in operating systems development. One approach is that of supplying user-desired features by defining the primitives available to user programs first, thereafter defining lower and lower levels of primitives within the operating system until the zero-level "nucleus" is reached. The reverse of this "top down" procedure is the "bottom up" approach, in which the lowest primitives are defined first. Since the thesis question itself involves a specific constraint on the lowest level of primitives and only general constraints on the highest, the latter technique was adopted in this study. Clearly, the two methods are complementary, and neither may be employed without regard for the other.

2. Ring Structures

One design goal deemed paramount from the beginning of the project was the innovation of a multiprocessor structure which has no optimal number of processors implicit in the structure itself. Various abstract models were considered, the first of which was a "ring" of processors. Such a system could be implemented in at least two ways. One technique is the "daisy chain." Each processor passes messages along to its neighbor, under this arrangement, and the receiving process (or its proxy, if it is currently inactive) ends the chain in each case. The rate at which messages move about in this system would be unnecessarily slow, since each processor must pause from productive computing for each message being passed through it. The more processors in the daisy chain, the greater the number of processors which must be "traversed" by an ever-greater number of messages. Another, more efficient means of implementing the ring structure is to provide each processor

a "smart" interface with a ring of fast shift registers. Here, each interface can perform the task of address recognition on behalf of its respective processor. The feasibility of this approach has, in fact, been shown [Ref. 8]. The ability of this system to withstand unrestricted growth in the number of processors in the ring is not unrestricted. As the circumference of the ring grows, the increasing average distance between communicating pairs of processors would slow the cooperation of processes and could progressively interfere with overall system throughput.

3. Data Bus Structures

Another type of system is one which requires a physical data bus to tie the processors together and a "bus process" to control the data flow. The speed of data transfer in such a system could be quite rapid: on signal from the bus process, a single processor is given control of the bus to pass a message or block of data directly to another processor or peripheral device or the bus process, itself. Alternatively, a multiplexing system could be imposed, dividing the physical bus into many time-slice channels, assignable dynamically by the bus process. Questions of reliability aside, the problem arises of how many processors could efficiently be serviced: the capacity of the data channels is bounded by the bandwidth of the bus itself [Ref. 1, p. 131]. Multiple bus lines are a means of expanding the bandwidth of a physical data bus, but this escape is in the direction of extreme complexity of hardware, if the bus is still to work as a coherent unity. Given that a system has K processors, the successive extensions of that system to one having K+1 processors, all sharing a common set of physical buses, will eventually require hardware modification or replacement of the original processors. In any case, the sophistication of the software and/or hardware necessary to implement a bus for over one hundred active processors would be impressive indeed. In order to extend such a bus structure indefinitely, some

means of decentralizing the bus workload would have to be found. Further study of the bus model was abandoned, since the need for bus control and the need for workload distribution were seen as competing goals.

3. Tree Structures

Attention was shifted to a "tree-structured" system. The immediate prospect of recursion in this design model prompted a closer look at modularity and component standardization as system goals. The result expressed in this proposal is a system based on a single hardware cluster, called a Processing Module (PM), to which may be appended certain accessories, such as bulk memories or user interfaces, without need for modification of the hardware or the software. These PM's are plug-connectable with each other in such a manner as to automatically extend the system in the form of a graph-theoretic tree.

In a tree-structured system, control functions can be processed recursively: each node is under the control of one higher node and in turn is the sole control for zero or more other nodes (up to a fixed limit). Thus, the root node of the whole system need only be aware of which branch under its control is responsible for a given process. That branch is a subtree whose root node knows which branch under its control holds the process in question. Ultimately, since the system is finite and contains no loops, a node is found which has no branches. This node must, therefore, be the location of the process. No higher node in the structure need be aware of this exact assignment, a fact which allows considerable economy in table space and lookup time at any given node. By extending these location tables at each node to include the priority of the process and by assigning the lowest priority to the null process (ie., the processor is free), the task of processor allocation is manageable on a recursive basis, as well.

The following section deals primarily with the hardware requirements of such a tree system, starting with

the Processing Module. Once the "machine" has been described, various aspects of an operating system are discussed under the heading of SOFTWARE STRUCTURE. Finally, a description titled SYSTEM OPERATION covers the behavior of hardware and software acting together.

Hereafter the proposed system will, for convenience, be refered to as TREE.

B. TREE HARDWARE STRUCTURE

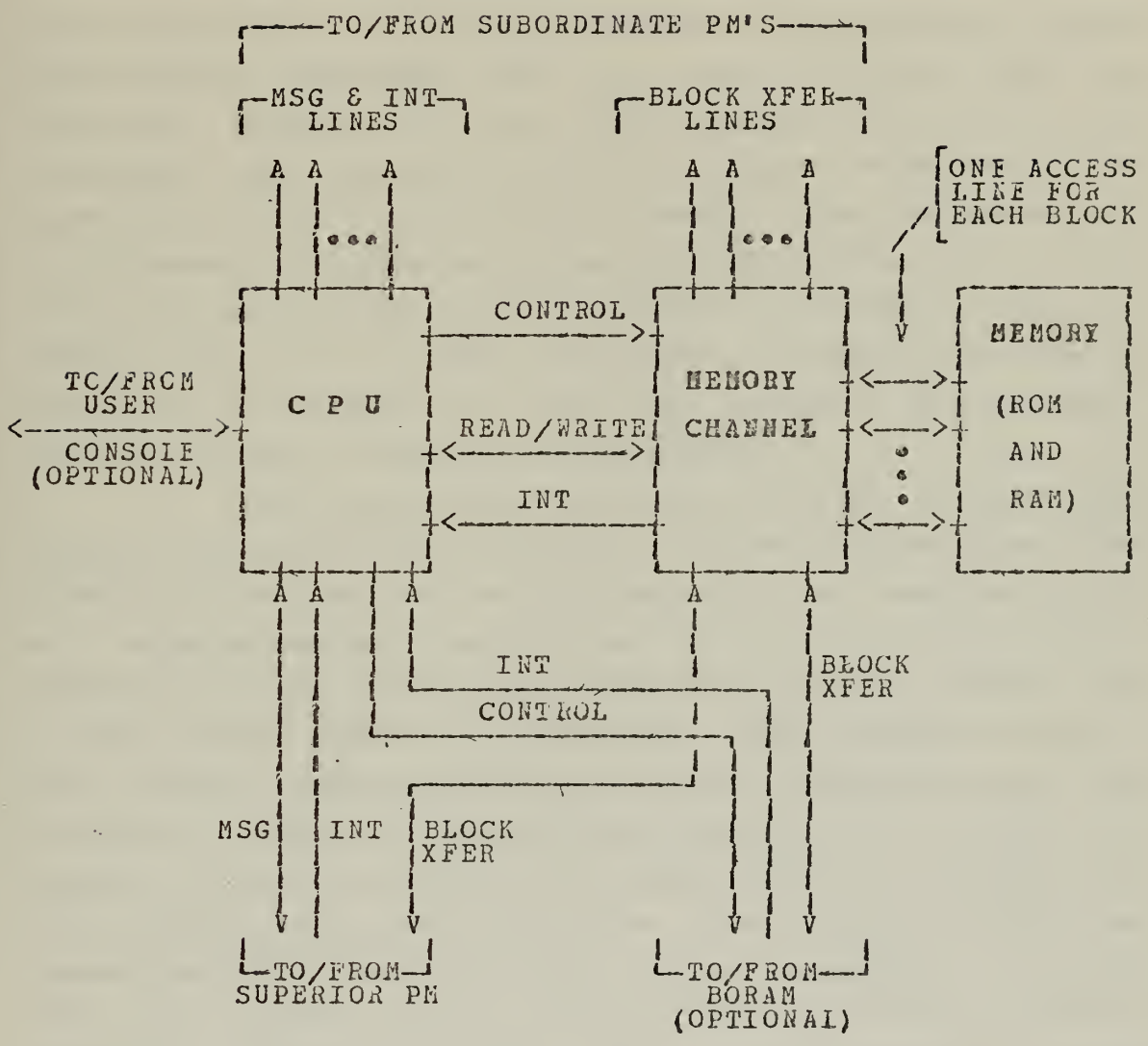
1. Processing Module Design

a. Central Processing Unit

The Central Processing Unit (CPU) is envisioned as a processor of arbitrary computing power. The word length and the capabilities of the instruction set are parameters which have no direct bearing on the feasibility of the system. Within limits, the power of the CPU could differ from one PM to another, or, more practically, from one branch to another. As long as the internal workings of each CPU are invisible to the remainder of the system, standardization of the CPU's themselves is not necessary. Strict adherence to the message/interrupt formats, which define the bounds between processors, guarantees this isolation. Variability in the computing power of the CPU offers the prospect of technical improvements without loss of compatibility with earlier TREE systems.

Figure 1 shows the CPU section of the PM in relation to other components. The apparent multitude of connections with the CPU are actually of only two types: interrupt lines and message/control lines. Interrupt lines arrive at the CPU from three sources. One is from the higher (controlling) PM; another is from the Memory Channel (page-fault signals); the third is from the (optional) bulk storage device. An internal clock can also generate interrupts. Interrupt lines also depart the CPU to as many subordinate PM's as the CPU design will allow. For the outbound interrupts, provision of externally-accessible

Figure 1. The Processing Module



flip-flops, one for each possible subordinate PM is necessary. These in turn are an addressable array to the CPU's logic and instruction set. Inbound interrupts are

subject to a priority scheme handled in the CPU hardware.

Message/control lines are tied to similar flip-flop arrays called the CPU software either in response to interrupts or in the normal course of process communication and memory access. They are all addressable internally as an extension of the CPU's memory space.

b. Memory Channel

The Memory Channel (MC) is a hardware processor which performs two distinct functions. In one role it maps relocatable addresses sent by the CPU into run-time addresses useable by the PM's memory. A vector of registers into which the hardware indexes is one means of providing the swift translations needed. The registers can be altered in response to control information sent by the CPU (when the segment control process is active). When an address maps to a page not held in memory, MC hardware generates an interupt to the CPU (causing the segment control process to become active again).

The other activity of the MC is as a switching network for block data transfers. Block transfer lines are provided in and out of the MC comparable to the interrupt and message/control connections of the CPU. Also, each independent memory block is accessible by this network as is the bulk memory, if present. Under periodic control from the CPU, the MC establishes access routes between the superior PM and any one of the memory blocks, between the superior PM and any of the subordinate PM's, or between the bulk memory and a subordinate PM. The MC only enables the connection directed by the CPU. The assumption is made that the result of any series of such connections within the total system is to allow block transfer between a bulk memory device and a memory block elsewhere in the system.

c. Memory (ROM and RAM)

The memory provided in the PM is broken down into independently-accessible blocks. The size of these blocks is arbitrary, the major constraint being that all

blocks in the system be of equal size. Some of these blocks are Read-Only Memory (ROM). These blocks provide for immediate start-up of the system by storing permanent copies of the minimum-system software. The remainder of memory is Random-Access Memory (RAM). The particular technology used to implement RAM is not important in this context. Various features could be designed into these blocks which, while not being necessary to the system, per se, would increase its power overall. For example, it would be valuable to be able to cause any of the memory blocks to input or output its entire contents at high speed and in word-sequential order via its block transfer line. This feature would allow rapid memory-to-memory transfers without need for mediating processors. Another desirable feature would be a block-level file protection system wherein access criteria are defined by the owner process in part of the block's memory space. All processes not permitted access by the owner-defined code are locked out no matter where that block is loaded in the system. By extension, the access code could be used to encrypt the entire contents of the block as it is being transferred out to bulk storage or to another memory. Generation and reduction of CRC parity-check codes are similar possibilities in this transfer process [Ref. 14, p. 16].

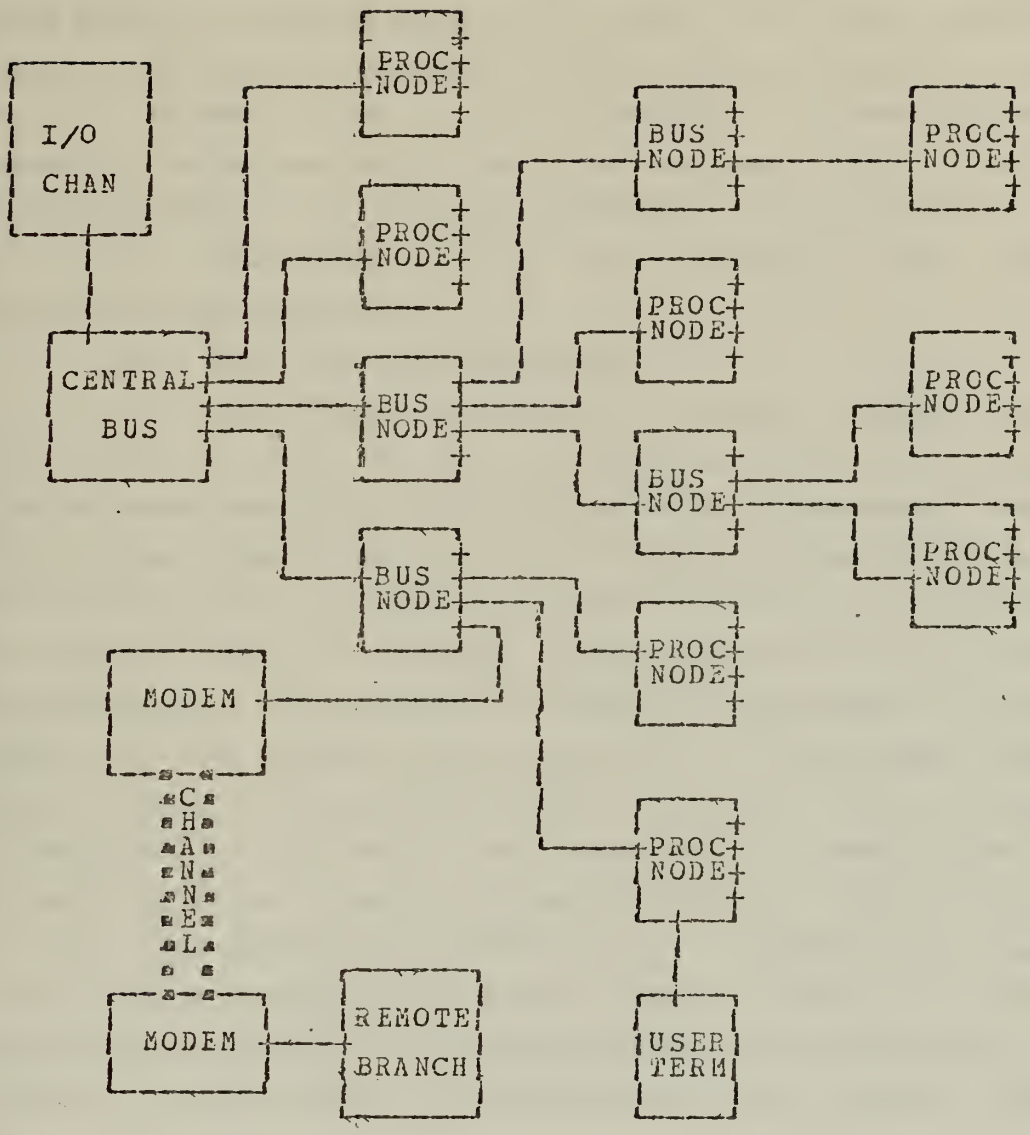
2. Module Interfacing

a. Data Lines and Channels

Figure 2 (page 24) shows a possible configuration of the TREE System. In the figure, the Central Bus, the Proc (Processing) Nodes and the Bus Nodes are all PM's, each operating according to its relative position in the structure.

The physical connections required between PM's in close juxtaposition can be achieved by printed circuitry or by flexible wire sets. The low Transistor-to-Transistor

Figure 2. A TREE Configuration



Logic levels (TTL) used internally are completely sufficient for short-distance inter-PM communications. The connection of any branch of the system (whether a single PM or a substantial sub-structure of PM's) to its controlling superior PM can be extended to any distance desired through use of MODEM's and a data channel. Since computation proceeds asynchronously in the various PM's and the message/control and interupt structure are designed to allow for this independence, the data channel has minimal constraints placed on it.

b. Bulk Storage Interface

Bulk "block-oriented" storage (BORAM) can be connected to the PM on an optional basis. The operating system described under SOFTWARE STRUCTURE presumes that all PM's have some BORAM connected except those which have no subordinate PM's. This arrangement allows the superior PM in every case to control bulk storage for its immediate subordinates. It could prove more advantageous to provide BORAM at all levels and to all PM's, in the event that the BORAM activity is excessive and interferes with processor throughput. (It would be necessary to design the Memory Channel to allow transfer from its attached BORAM to any of its own Memory Blocks in this case.) Schemes for sharing a single large BORAM system by all PM's within a level or branch are readily imaginable but detract from the strict recursive structure being offered here as a general model.

c. User Interface

Another optional connection provided in the PM design enables conversational user terminals to be interfaced anywhere in the system structure. In practice, a single branch of the system, rather than a scattering of PM's, would probably be assigned to user terminals. Figure 2 shows a single terminal for an entire system. Any of the Processing Nodes shown could have a terminal added. Input-output for each terminal is programmable as a high-level process to be carried out by the PM whenever its

associated terminal is on line. The user thus has the benefit of a "smart" terminal, ie., one which handles I/O, edit functions, and some file control without appeal to the larger system. The system, on the other hand, retains access to all its processors and is able to use any or all of the PM's connected to terminals on a priority basis if the terminals are on line. Each terminal-connected PM automatically reverts to general system use whenever the user logs cut or when terminal power is cut.

3. System Structure

a. Bus and Processing Nodes

In terms of hardware, all PM's in the system are identical. The tree-structured interconnection of these modules creates a natural division of labor which is conceptually useful. The "leaves" of the system tree, those PM's which have no subordinates, may be thought of as the "Processing Nodes," whereas all PM's superior to these leaf-PM's are involved with system management, especially communications, and may be regarded collectively as "Bus Nodes." The system is essentially a hierarchy of Bus Nodes, branching outward from a single Bus Node and terminating in all cases with Processing Nodes.

b. System Input-Output

The Bus Node at the apex of the hierarchy (the Central Bus) is left with a set of message/control lines, an inbound interrupt line, and a block-transfer line which are part of the PM standard design but which, by definition, lead to no higher bus. The input-output channel is given access via this central interface with the system. High-speed I/O devices are multiplexed by the I/O channel, which is under the control of the Central Bus. The interrupt line allows the channel to proceed independently and notify the Central Bus when an assigned I/O task is complete. The block transfer line allows the channel to access Central Bus memory to fetch or overlay on a block-at-a-time basis.

c. BNF System Specification

Emphasis must be placed on the recursive nature of the system structure. Any PM in the system which is serving as a Processing Node can at once be changed into a Bus Node by connecting one or more new PM's and a BORAM. The degenerate case of a system consisting of exactly one PM connected to an I/O channel is a feasible configuration, even though multiprocessing is not possible. Inspection of this uni-processor system reveals that it is not very different from a "conventional" uni-processing system. The design of the PM is essentially a generalization of "classic" system design: the CPU has its own main memory; main memory is backed up by bulk memory (disk, drum, or tape); input-output has independent access to memory; and the overall hardware implements an interrupt structure. Input-output is the link between a conventional processor and the TREE system. Each input and output port or group of ports is assigned meaning by the software and hardware acting together. Part of that meaning is the hierarchical relationship which all PM's share.

In light of the recursiveness of the structure, it is possible to specify concisely the rules for structuring a TREE system. To achieve this description, it is useful to adopt a notation already in wide use for the description of context-free (tree-structured) computer languages: Backus-Naur Form (BNF). Since BNF is descriptive of possible linear arrangements of symbols rather than of the possible tree structures used to arrive at those symbol strings, the following adjustment is necessary. The concatenation of two variables (represented by "•") is interpreted to mean that the first is connected to the second. Should the second variable be a list of variables, the meaning intended is that the first is connected separately to each member of the list. Variables which are lists are defined as such, using list notation.

Table II (page 27) gives the productions for

TREE. The configuration illustrated in Figure 2 may be checked by successive applications of these rules. For example, rules two and three disallow the connection of a new <PM> onto any <PM> which has a <TERMINAL> attached already. Rule four implies that for a <P-NODE> to become a <B-NODE>, the <TERMINAL>, if present, must be removed and a <BORAM> added. Any legal TREE configuration may be generated using these rules, as well. Since some variables appear in the rules connected to a non-empty list of variables, list notation should be used to linearly specify any resulting structure. The configuration depicted in Figure 2 may be represented by the following (abbreviated) statement:

<SYSTEM> =

<IOGROUP>•B•(P,P,B•(P,B•(P,P)),B•(P,PT,<REMOTENODE>)),

where B is a <B-NODE>, P is a <PM> and PT is a <PM>•<TERMINAL>.

Table II. BNF System Description

1. <SYSTEM> ::= <IOGROUP>•<BRANCH>
2. <BRANCH> ::= <P-NODE> | <B-NODE>•<BRANCHLIST> | <B-NODE>•<REMOTENODE>
3. <P-NODE> ::= <PM> | <PM>•<TERMINAL>
4. <B-NODE> ::= <BORAM>•<PM>
5. <BRANCHLIST> ::= (<BRANCH>) | <BRANCHLIST>U(<BRANCH>)
6. <BORAM> ::= <DISK> | <DRUM> | <BORAM>•<HCSTORE>
7. <IOGROUP> ::= <IOCHAN>•<DEVICELIST>
8. <DEVICELIST> ::= (<IODEVICE>) | <DEVICELIST>U(<IODEVICE>)
9. <REMOTENODE> ::= <MODEM>•<CHANNEL>•<MODEM>•<BRANCH>

Notes: a. BORAM: Block-Oriented Random Access Memory
b. HCSTORE: High-Capacity (on-line) Storage
c. PM: Processing Module
d. IOCHAN: Input-Output Channel
e. IODEVICE: card reader, line printer, etc.
f. MODEM: Modulation-Demodulation Unit
g. CHANNEL: Data Transmission Channel
h. •: Concatenation: "is connected to"
i. U: Union of two lists

C. SOFTWARE STRUCTURE

1. Programs vs. Processes

A very necessary distinction must be drawn between a software program and the process it controls. This fact is particularly true in TREE. A program written to implement a given process must be able to run on several processors at once. The same code on two different processors represents two distinct processes. This constraint is necessary in a system which treats processes as named individuals, which can generate messages requiring replies. If the originator of a message is not distinct from a programmatic twin, great confusion can result.

A process is not necessarily tied to its processor, although minimum-system processes are resident on their respective processors. Processes are generally allowed to migrate about the system. A message sent from one processor may have to receive a reply at another processor.

2. Message Primitives

The main problem with the extension of Hansen's communications primitives to multiprocessing is found in the handling of the associated message buffer pool. As originally formulated, each process draws upon this pool when initiating a communication to another process (up to a preset limit, to prevent "black sheep" processes from capturing the whole system). Provided that an answering process uses the same buffer for its reply as was sent to it, communicating processes have a mutual identification of a given communication: the name of the buffer itself. Buffers are thereafter returned to the common pool. For communications which require no reply, the buffer is returned at once to the pool.

The difficulty with implementing this buffer pool scheme for a large number of CPU's is that of memory

access.[1] One means of circumventing the necessity for central memory is to compromise on the buffer-pool concept itself. Instead of a single, central pool, each process is provided with an input queue of nominal length within its own memory. Whenever that length is exceeded, a scan of the queue is performed to determine if the sending process is over-represented already. If so, the offending process is notified and/or removed from the system. If not, the queue can be extended and notice of the overflow sent to a dump process for later analysis.

Identification of a communication, whenever two or more are active between two processes, is somewhat more troublesome but is resolved by the "naming" of messages by the initiating process. The number of simultaneous communications possible under this arrangement is limited either by the size of the name-space for which the initiating process has room or by the length of the name field in the message format.

3. System Hierarchy

All processes in the TREE system are members of a strict hierarchy, a concept first applied by Dijkstra in his design of the "THE" operating system [Ref. 7, p. 343]. Each successive level in the hierarchy provides a new degree of functional abstraction. Level zero implements the communications primitives suggested by Hansen. All higher-

---

[1] A system capable of emulating the ILLIAC IV, having a minimum of 65 processors, would have to provide some means of access for all processors to the main memory which would not degrade the memory cycle time of the processors individually. This requirement clearly exceeds current feasible technology. Memory would have to be continuously readable by all processors, similar to a large status board being continuously readable by all workers in an office space. While laser holography or some equally exotic medium may one day provide this kind of large-scale simultaneous access, a less direct approach must be devised for the interim generation of computer systems.

level processes are able to use these primitives without further concern for the means employed. Implementation of the primitives is performed by processes residing at each PM. Higher-level processes need only call one of these "system sub-routines" and wait for control to be returned. Messages received at a subordinate PM are accompanied by an interrupt. Each processor's level zero includes an interrupt-response process which can in turn load certain other processes in order to service a received interrupt. The interrupt structure is thus at the exclusive disposal of the zero-level processes. No process above level zero is permitted direct initiation of an interrupt, and all received interrupts are interpreted by zero-level processes. Included in level zero is a Message Queue Handler, whose job it is to add messages to process input queues, and a Process Locator, whose task is to pass the messages to a subordinate or the superior PM if the addressed processes are not locally held. Referring to Figure 2, a message generated in the Processing Node servicing the user terminal (lower right) and destined for a process unknown to that node would be passed to the superior Bus Node connected to it. If the addressed process is unknown to the Bus Node as well, the message would be passed up to the Central Bus. If the process exists, the Central Bus will know which of its subordinates is responsible for it and will route the message to that branch. The processor receiving the message from the Central Bus performs a similar search. Eventually, the addressee is found in some subordinate's local file of processes and the message is added to the input queue of that process.

Finally, level zero contains a process Creation/Removal process (to be explained under SYSTEM OPERATION). Because the zero level behaves as a relatively self-sufficient society, it is convenient and proper to refer to it as the "nucleus." The nucleus is identical on

all PM's in the system. Hansen's definition of an operating system nucleus, allowing for the context of uni-processor systems in which it was written, is compatible with the one being presented:

> "Multiprogramming and communication between internal and external processes are coordinated by the system nucleus -an interrupt response program with complete control of input/output, storage protection, and the interrupt system. We do not regard the system nucleus as an independent process, but rather as a software extension of the hardware structure, which makes the computer more attractive for multiprogramming. Its function is to implement our process concept and primitives that processes can invoke to create and control other processes and communicate with them." [Ref. 10, p. 239]

The remaining levels above the nucleus provide successive degrees of abstraction, permitting user-level processes use of as powerful a set of primitives as possible. The procedure of assigning processes to levels admits a good deal of variation. More important in the present context is the assurance that the requirements of the tree-structured relation among PM's are compatible with the requirements of each PM's operating system. In analyzing this interaction in the section on SYSTEM OPERATION, the hierarchy assignments shown in Table III (page 30) will be assumed. As indicated in the table, levels zero through four are "minimum system" processes whose software is kept in each PM's ROM. All other processes must be fetched from backing stores on demand.

4. File Protection

As pointed out earlier, very secure file protection is obtainable by performing access checks within the RAM (hardware) block unit, based on a header of information stored with the contents of the block. Another approach is to make only the access header available to the Memory Channel Control process whenever the block is first placed in RAM. The MC Control process can then include this information with the data it supplies to the MC translation registers. Thereafter, each access which maps to that block

is checked for proper credentials and an interrupt generated whenever an illicit read or write is attempted. Within this framework a protection code could be developed to allow the user or process designer to specify type of access for any subset of process levels and/or system users. The security available through such a protection system is dependent upon the loading of PM memory with descrete blocks from BORAM or the I/O Channel. Any skew of the Memory Blocks and the data blocks being loaded would nullify this protection, since words from a protected data block would overlap onto another Memory Block not necessarily having the same header information.

Table III. A Sample Hierarchy

LEVEL	NOTES	TASKS ASSIGNED
0	A	Process Communications; Process Creation and Removal
1	B	Process Scheduler; Clock Process
2	B	Segment Control (Memory and Bulk Storage allocation)
3	B	I/O Buffering; Memory Channel Control; BORAM Control
4	B	Bus Node Process
5	C	System Operator; Library Routines; User Processes

Notes: A: Nucleus processes
B: Created, non-transferable processes
A&B: Minimum System (stored in ROM)
C: Transferable processes

D. SYSTEM OPERATION

1. Steady-State Operation

a. Process Creation and Removal

System operation may be viewed generally as the creation, control and removal of processes. The nuclei of the system act independently, but collectively, to achieve this abstraction. As the major means of communication among higher-level processes, they tend to be the focal point of control activity. Creation and removal of other processes is

the most powerful form of control of all and is retained as a nucleus function useable globally via primitives, in order that the credentials of each process attempting their use may be checked.

It is important to note that the nucleus processes are not able to communicate with each other via the primitives which they implement. Nor is there any need for the zero-level processes to converse with other levels or with their counterparts on other processors. By the same logic, the entire nucleus exists *a priori* and is non-destructable: since the creation and destruction of processes is controlled by the nucleus, operation of this function upon the nucleus itself could lead to confusion and deadlock.

The nucleus creates and controls all higher-level processes. This relationship serves to avoid ambiguities which could result in deadlock. For example, if, in response to an interrupt, the nucleus starts creating a higher-level process, the receipt of another interrupt while response to the original one is under way poses a very real problem, but a controllable one. Once a process has been created, it may be interrupted and stored. If necessary, the software program on which it was based may be used to create a new programmatically identical process. Creation of a process is a brief sequence which involves setting aside memory space for the input message queue of the new process, adding its name to a local list of known processes and notifying the superior bus that a process by that name now exists. The Process Scheduler, using its table of existing processes, assures that all interrupted processes are ultimately re-activated, passed to another processor, or removed from the system. The problem posed by the above example reduces, then, to assuring that the creation process is not interrupted.

b. Process Clocking

The Creation/Removal process is not the only one

requiring protection from untimely interruption. The other nucleus processes are equally vulnerable. Some means of "clocking" each process with respect to the one proceeding on the superior processor must be present, ie., some temporal interlock which can control the competition of these separate processes [Ref. 11, p. 13]. Dijkstra's "mutex" operators [Ref. 7, p. 345] provide a means of disabling interrupts while a nucleus process is active. Since the nucleus processes are programmatically very brief, the delay in servicing a pending interrupt is not serious. There are four sources for interrupts: the Memory Channel, BORAM, PM Clock and the Superior PM (I/O Channel, in the case of the Central Bus). Granting service priority in just that order, Memory Channel first, assures that a hyperactive superior cannot overload a subordinate.

The ability of a user process to call for creation of other processes facilitates another method of process clocking, one which avoids the need for Dijkstra P and V operators in the instruction set available to general users. Whenever a user-defined "community" of processes is created, ie., a group of processes which communicate at least in part via a common set of variables, critical section problems can be averted by simultaneously creating a custodial process which performs all critical section operations from its input queue and returns advice by message. Calls for the creation of these communities, when sufficiently well-defined, can be performed by system compilers directly from such higher-level language constructs as FORK and JOIN or DOTOGETHER.

Clocking may also be accomplished using the message structure in problems involving dependent-element grid processing, such as heat-transfer through a solid. Each point on the grid is represented by its own process. Rather than providing a central body of variables and a custodial process, here each process can be required to "broadcast" significant data (to its orthogonal neighbors,

for instance) after each iteration. Again, system compilers may prove extremely well-suited to forming such communities.

c. Multiprogramming

Each PM maintains a table of processes known to it. PM's acting as Bus Nodes therefore include in this table the identity and (relative) location of processes existing on all subordinate processors. Periodically, the Process Scheduler is activated in each PM by the Clock process. A review is made of the processes locally active for selection of the next process to proceed. For the Bus Nodes, the Bus Node process has the highest priority, since it must scan the message input ports from subordinate processors on a relatively frequent basis. Processing Nodes, having no subordinates to worry about, select from the entirety of their known-process table, running the Bus Node process only as a last resort as the "null" process.

Another task of the Process Scheduler is to review the distribution of processes assigned to subordinates and to initiate transfers from overloaded branches under its control. This computation must take into account that processes which belong to a process community (as previously defined) will run less efficiently if assigned to a single processor. (To avoid this situation, community processes must be assigned to separate processors initially and then given highest priority to remain there until removed from the system.) In general, multiprogramming is a "hyperprocess" consisting of the Process Scheduler and Processor Workload Control. The Scheduler has the power to replace a process which is blocked (by a page fault, for instance) by another which is able to run, even if the only replacement is the null process. Workload Control allows processes to "float" about the system, thereby extending multiprogramming across Module boundries to include the entire system.

d. Error Detection and Debug Facilities

Perhaps the most difficult aspect to analyze of

any system, hypothetical or otherwise, is its ability to analyze itself. Error detection facilities must be considered thoroughly in the design of any modern system, as must be its debug facilities, if disasterous development and maintenance costs are to be avoided. Of all the design features important to the success of these efforts, modularity of system design is critical. [Ref. 15, p. 548]. Without well-defined delineations among the many functional parts and levels of control, the detection and localization of error conditions boarders on the impossible. TREE attempts not only to offer modularity but also to clearly define the methods of cooperation among processes allowing swifter isolation of unanticipated interactions. Fault detection can be implemented in several areas of concern: unauthorized file-entry attempts; unauthorized process creation/removal and communications attempts; inter-processor data parity checks; and processor deadlocks (detected by periodic assignment of test processes to each processor). The message primitives offer a simple means of informing dump processes of the time and location of detected errors without the need to load a new process on the (possibly) defective processor.

2. Non-Steady-State Conditions

a. System Initiation

The Minimum System for TREE is stored on ROM in each of the PM's system-wide. The transition from a "cold" machine to a functioning system is accomplished by a Bootstrap Routine which is also held in ROM but is never made into a process. When power is applied to the processor, the operator is allowed to reset the relocation registers of the system Memory Channels and force a fetch of the first Bootstrap instruction from ROM. Thereafter, the Bootstrap is able to issue the control sequence necessary to initialize the MC registers not associated with the fetching of its own program, followed by an initialization of tables for the nucleus. Its final act is to yield to the nucleus

by calling on the creation primitive to create the Process Scheduler.

Once active, the Scheduler is programmed to react to the absence of other Minimum System processes by successively calling for their creation. Eventually the Bus Node process is activated and is able to determine whether it has subordinates. If there are none, the Scheduler is notified to reduce the priority of the Bus Node process to the lowest possible. At this time, the processors are fully operational and able to accept processing assignments by unlocking user consoles and enalbing the central I/O Channel.

b. Degradation and Maintenance

Systems must be able to adapt themselves to progressively worsening component failure without undue side effects. Clearly, in a tree-structured system, the complete malfunctioning of a particular node is less tolerable the closer it is to the root node. Failure of the root node (Central Bus) itself is serious, indeed, but not fatal to the entire system. The relative independence of each node serves to sustain and protect existing processes while the offending processor is reloaded or replaced. With due attention to this possibility in the design of PM hardware and software, removal and replacement of a module could be effected without shutting down the system. At the cost of generality of module design, parallel redundancy can be built into the Central Bus, allowing it to simulate the behavior of the standard PM's while providing needed extra reliability.

c. Reconfiguration

As noted in the previous section, the need to close down the entire system while physical connections are being made or broken is not absolute. In terms of software, the behavior of the Bus Node process facilitates adaptation to such instant alterations: a Processing Node can become a Bus Node as soon as it recognizes a meaningful input from

one of its previously dormant subordinate-communications lines. The node may then upgrade the priority of its Bus Node process and transfer its processing load tc the new processor (or branch). Conversely, if all transferable processes are withdrawn from a Bus Node's subortinates, the subordinates may then be unplugged, causing the Bus Ncde to revert tc Processing-Node status. Any branch which is unplugged prior to being relieved of its transferable processes continues to function normally, provided it still has electrical power. The demands placed upon operating-system logic to withstand such divisions might be great or might be quite trivial, but the prospect of computer systems which are allowed to "grow" and "divide" may well be worth investigation.

III. CONCLUSIONS

A. SUFFICIENCY OF THE BUS SYSTEM

The ultimate question considered in this study has been the feasibility of a multiprocessing system having communications as its focal process. The nucleus of the TREE system, together with the functional adaptiveness of each module, may be thought of as a "smart bus," over which virtually all system processes may communicate. The structure of the system permits distribution of control, avoiding the bottleneck of a single bus processor, without the necessity for complex data-transfer hardware. This simplicity in hardware works to the benefit of system reliability. The uniformity of each module's operating system provides a very real measure of software reliability and maintainability.

B. GENERAL-PURPOSE CAPABILITIES

The versatility of a large system is its strongest justification. The growing need for powerful multiprocessing systems has prompted some offerings which are less than general in capability, representing a departure from the mainstream of computing development. The proposed system enables users to create processes dynamically and to define their interaction while, at the same time, providing sufficient processing power to assure high throughput.

C. VARIABLE PROCESSING POWER

A recursively-expandable system design is offered as a solution to the problem of ever-increasing demands on processing power. From a practical point of view, the capability emerges for a computing center to adjust to its volume of work by smaller increments than is presently possible. This adjustment could be downward or upwardward.

D. TECHNOLOGICAL FEASIBILITY

The design of the Memory Channel section of the Processing Module has been stated in broad terms and could require considerable technical development to make it a reality. Of the Channel's two functions, dynamic address translation offers less of a problem. For the remaining function, it is not clear that a switching network can economically be devised capable of variously interconnecting BORAM and the superior Bus Node to an arbitrary number of subordinate nodes and Memory Blocks. As a complex enable circuit, the number of gates required might be rather large. Granting that this problem can be resolved, it should be noted that all other components of the Processing Module are conventional in terms of present or expected LSI technology.

BIBLIOGRAPHY

1. Abramson, N., Information Theory and Coding, McGraw-Hill, p. 93-135.

2. Aspinall, D., Kinniment, D. J., and Edwards, D. B. G., "Associative Memories in Large Computer Systems," Information Processing 68, v. 1, p. 796-800, North Holland, 1969.

3. Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, v. 5, p. 31-80, March 1973.

4. Control Data Corporation Report PDJ #33, STAR Operating System Concepts, 42 pp., 18 September 1969.

5. Balzer, R. M., "PORTS -A Method for Dynamic Interprogram Communication and Job Control," AFIPS Proceedings, SJCC, v. 38, p. 485-489, AFIPS Press, 1971.

6. Denning, P. J., "Virtual Memory," Computing Surveys, v. 2, p. 153-189, September, 1970.

7. Dijkstra, E. W., "The Structure of the THE-Multiprogramming System," Communications of the ACM, v. 11, p. 341-346, May 1968.

8. Farber, D., "Data Ring Oriented Computer Networks," Courant Computer Science Symposium 3, Rustin, R., ed, p. 79-93, Prentice-Hall, 1971.

9. Gosden, J. A., "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-processor Computers," AFIPS Proceedings, FJCC, v.29, p. 651-660, Spartan Books, 1966.

10. Hansen, P. B., "The Nucleus of a Multiprogramming System," Communications of the ACM, v. 13, p. 238-250, 1973.

11. Horning, J. J. and Randell, B., "Process Structuring," Computing Surveys, v. 5, p. 6-30, March 1973.

12. MCS-8 Micro Computer Set Users Manual, rev. 3, Intel Corporation, March 1973.

13. Shaw, A. C., The Logical Design of Operating Systems, (Draft dated September, 1971) reprinted at the Naval Postgraduate School with permission of the author, p. 4.22-4.33, 1972.

14. University of Michigan Technical Report 20, Topics in Computer Communications Systems, by D. L. Mills, p. 13-21, May 1969.

15. Wichmann, B. A., "A Modular Operating System," Information Processing 68, v. 1, p. 548-556, North Holland, 1969.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Chairman, Computer Science Group, Code 72 Naval Postgraduate School Monterey, California 93940	1
4.	Asst Professor G. A. Kildall, Code 72Kd Computer Science Group Naval Postgraduate School Monterey, California 93940	1
5.	Asst Professor R. H. Brubaker, Jr., Code 72Bh Computer Science Group Naval Postgraduate School Monterey, California 93940	1
6.	LT Richard J. Goodwin, USN 235 Alder Street Pacific Grove, California 93950	1

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Design for a Distributed-Control Multiple-Processor Computer System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; December, 1973 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard James Goodwin		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December, 1973 13. NUMBER OF PAGES 42
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Multiprocessing; multiprogramming; modular design; operating system; process concept; process communication; process hierarchy; process creation; process removal; process clocking; process interaction; recursion; message primitives; file protection; error detection.

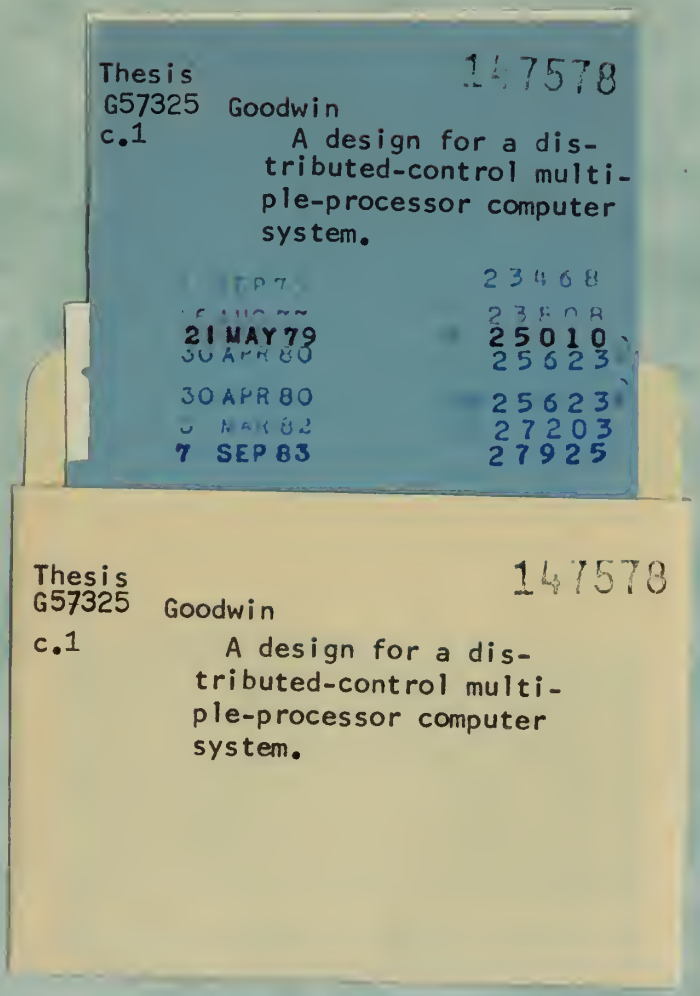
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A tree-structured multiprocessing system design is proposed in which process communication is the primary link between processors. A hardware cluster, called a Processing Module, is proposed as the basic structural component. These modules literally "plug together" to form a system of arbitrary size. Each module has its own memory and runs its own hierarchically-structured operating system, the nucleus of which implements P. B. Hansen's communications primitives along with process creation and removal. Workload scheduling and

DD FORM 1 JAN 73 1473 (Page 1) EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

process location are performed recursively in the system's tree structure. Multiprogramming is implemented system-wide, allowing processes to migrate away from overloaded modules. It is argued that the resulting system would be truly general-purpose and is subject to no limit on its size and consequent computing power.



Thesis
G57325
c.1
147578
Goodwin
A design for a distributed-control multiple-processor computer system.